DEVELOPER MANUAL

WP2.2 MPI Code Debugging and Verification (MARMOT)

Document Filename:	**CG2.2-v0.1-UST001-MARMOTDeveloperManual-v1.1.12.doc**
Work package:	**WP2.2 MPI Code Debugging and Verification (MARMOT)**
Partner(s):	**USTUTT, CSIC**
Lead Partner:	**USTUTT**
Config ID:	**CG2.2-v0.1-UST001-MARMOTDeveloperManual-v1.1.12**
Document classification:	**PUBLIC**

Abstract:
This is the developer manual for the MPI analysis and checking tool MARMOT.

Document Log

Version	Date	Summary of changes	Author
0.1	20/12/2004	First draft. Section 2.3 is not finished yet.	Bettina Krammer
	26/01/2005	Verified by the QE	Robert Pajak

CONTENTS

LIST OF FIGURES

COPYRIGHT NOTICE

This research is partly funded by the European Commission IST-2001-32243 Project “CrossGrid”.

INTRODUCTION

1.1. INTRODUCTION TO MPI STANDARD

The Message Passing standard MPI has been passed to increase the portability of message passing applications, by adopting standardized interfaces for the communication. Among the key aspects of MPI are:

- Definition of C and Fortran77 API (MPI-1) respectively C++ and Fortran90 API (MPI-2).
- Additionally to the transfer of simple byte-streams, MPI introduces the concept of *Datatypes* for the communication, which gives the user the possibility to pass heterogeneity aspects in Grid-computing (partially) to the MPI library.
- MPI defines a wide range of point-to-point operations, giving the user the flexibility to choose the one fitting best to the application. Among the supported methods are blocking and non-blocking operations (identified by so-called *Requests)*, buffered and synchronous send modes, and persistent communication.
- MPI introduces a flexible grouping concept based on *Communicators*, providing a pre-defined group called *MPI_COMM_WORLD*, which includes all processes started.
- Additionally to point-to-point operations, MPI abstracts several typical communication patterns to so-called *Collective Operations*, giving the MPI library the possibility to optimize these communication patterns.

The MPI standard has been extended in 1997 by the MPI-2 document, which defines additional functionality (e.g. one-sided communication, dynamic process management and parallel file I/O). The MPI 1.2 standard is described in the MPI 2.0 document. It is basically the 1.1 version with a number of clarifications and bug fixes. Since the number of full MPI-2 implementations is currently very limited, e.g. there is no freely available implementation for Linux, the version 2.0 of the standard has not yet been widely adopted by the user community. Therefore, the verification tool presented in this document will be limited to the versions 1.1 and 1.2 of the standard.

1.2. INTRODUCTION TO MPI CHECKING TOOLS

The Message Passing Interface (MPI) is a widely used standard for writing parallel programs. However, the MPI-1.2 standard, with its 129 calls, has a size and complexity that makes it possible to use the MPI API incorrectly. According to our experience there are several reasons for this:

- Developers do not only have to face all the problems that occur in serial programming. In addition, parallel applications get more and more complex and especially with the introduction of optimisations like the use of non-blocking communication also more error prone.
- MPI programs do not always behave deterministically. Deadlocks or race conditions may appear, depending on the platform environment or on the MPI implementation.
- The MPI standard leaves many decisions to the implementation, e.g. whether or not a standard communication is blocking. This implementation-defined behaviour may cause problems when porting an application from one platform to another, for example, when porting an application from a local platform to the CrossGrid testbed.

Debugging MPI programs has been addressed in various ways. The different solutions can be roughly grouped in four different approaches: classical debuggers, special MPI libraries and other tools that may perform a run-time or post-mortem analysis.

- Classical debuggers have been extended to address MPI programs. This is done by attaching the debugger to all processes of the MPI program. There are many parallel debuggers, among them the very well-known commercial debugger Totalview. The freely available debugger gdb has currently no support for MPI, however, it may be used as a back-end debugger in conjunction with a front-end that supports MPI, e.g. mpigdb. Another example of such an approach is the commercial debugger DDT by streamline computing, or the non-freely available p2d2.

- The second approach is to provide a debug version of the MPI library (e.g. mpich). This version is not only used to catch internal errors in the MPI library, but it also detects some incorrect usage of MPI by the user, e.g. a type mismatch of sending and receiving messages.
- Another possibility is to develop tools dedicated to finding problems within MPI applications. At present, three different message-checking tools are under active development: MPI-CHECK, Umpire and MARMOT. MPI-CHECK is currently restricted to Fortran code and performs argument type checking or finds problems like deadlocks. Like MARMOT, Umpire uses the profiling interface. Unfortunately, Umpire is not freely available. These three tools all perform their analysis at runtime.
- The fourth approach is to collect all information on MPI calls in a trace file, which can be analysed by a separate tool after program execution. A disadvantage with this approach is that such a trace file may be very large. However, the main problem is guaranteeing that the trace file is written in the presence of MPI errors, because the behaviour after an MPI error is implementation defined.

1.3. INTRODUCTION TO MPI CHECKING TOOL MARMOT

The MPI verification tool is provided as a library that implements the MPI standard. More precisely, it uses the MPI profiling interface to intercept every MPI call made by the application. The MPI profiling interface gives the user the possibility to replace MPI calls by routines with different functionality, and provides a second, name shifted version of all MPI routines. The verification tool uses this interface to check the MPI calls for consistency and correctness before passing it to the real MPI library for processing. The system will therefore consist of two major parts:

1. The implementation of the API defined in the MPI standard. As required by the MPI standard this API should be a library. The MPI calls of the applications are caught and passed to the core MPI verification tool through this library.
2. The core MPI verification tool. This tool provides the real verification and debugging functionality. It is implemented in C++. A detailed description is provided below. Since two language bindings are defined by the MPI standard, these two language bindings have to be implemented (C and Fortran). The C interface is implemented in C++, but providing a C interface to external programs, i.e. implementing the C binding of MPI. The Fortran interface is implemented as a wrapper to the C interface. Thus we avoid implementing the core functionality twice. Special care has to be taken for the implementation of checks that are different between the C and Fortran implementation, for example checks concerning datatypes. Some of the checks implemented in MARMOT are also performed by MPI implementations. However, such checks are not required by the MPI standard, therefore an implementation of the MPI standard does not need to include these tests and, for the sake of performance, expensive checks are usually not implemented.

1.4. ABBREVIATIONS AND ACRONYMS

MPI	Message Passing Interface
API	Application Programming Interface
MD	Migrating Desktop

1.5. REFERENCES AND SOURCE CODE

- MARMOT's source code can be found at
http://savannah.fzk.de/cgi-bin/viewcvs.cgi/crossgrid/crossgrid/wp2/wp2_2-verif/src/MARMOT/
- MARMOT's RPMs can be found at
http://gridportal.fzk.de/distribution/crossgrid/autobuilt/i386-rh7.3-gcc3.2.2/wp2/RPMS/
- The latest tag is currently v1.1.12.
- More information can be found at MARMOT's homepage
  http://www.hlrs.de/organization/tsc/projects/marmot
  especially on the publications site
  http://www.hlrs.de/organization/tsc/projects/marmot/pubs.html

IMPLEMENTATION STRUCTURE

MARMOT does not change the MPI library or the application. We use the profiling interface that is defined in the MPI standard 1.2, but probably not all users know this profiling interface or have ever used it. The idea of the profiling interface is simple: any MPI call `MPI_Foo` can also be invoked by `PMPI_Foo`, they are exactly the same. This allows users to define their own MPI routines, and that's also what MARMOT does: to put it in a simple way, we redefine `MPI_Foo` by

```
{
   do the MARMOT checks;
   PMPI_Foo;
}
```

Thus we check the MPI calls made by the application and pass them to the underlying MPI library. Marmot is just an addition to the MPI library, not a replacement.

MARMOT also has its own book-keeping of resources such as datatypes, groups, communicators, etc. This enables the tool to check if these resources are used in a correct way, for example if they are constructed, used and freed properly.

MARMOT requires an additional process (our debug server), which is responsible for tasks that require a global view, e.g. deadlock detection. Local tasks such as the verification of resources are performed on the client side. The additional process is caught automatically by the tool, the user only has to run the application with one additional process. In order to ensure that the additional debug process is transparent to the application, we map MPI_COMM_WORLD to a MARMOT communicator that contains only the application processes. Since all other communicators are derived from MPI_COMM_WORLD they will also automatically exclude the debug server process. The figure below shows a graphical representation of the way MARMOT works.

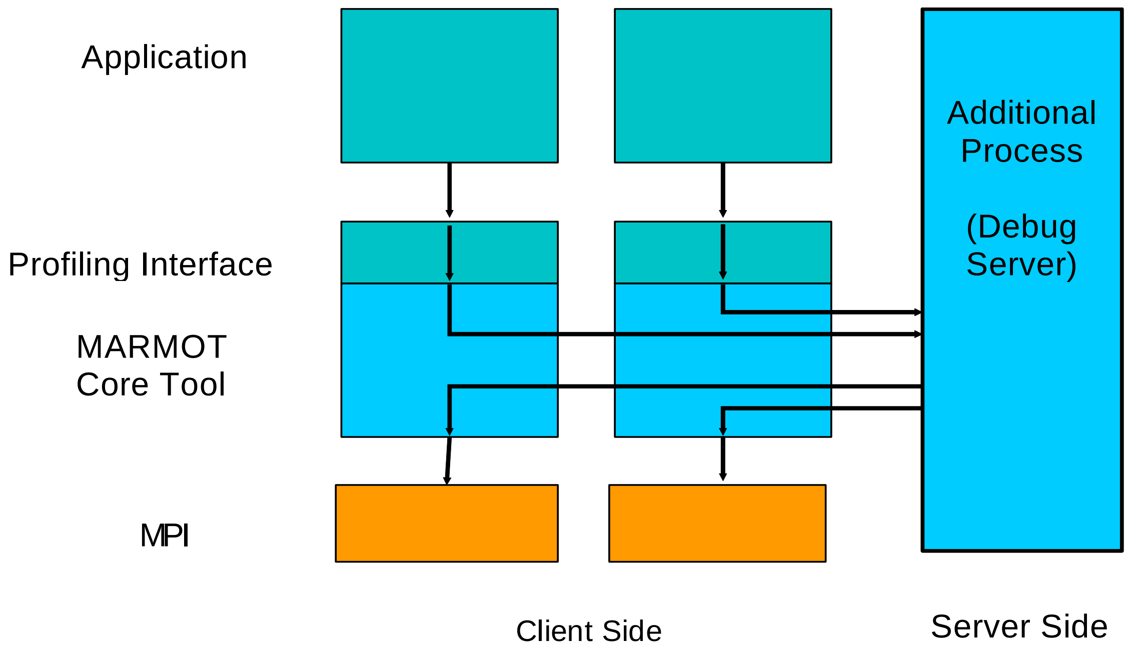

figure 1 Design of MARMOT

The output of MARMOT is a human readable log file. MARMOT's functionality includes:

- MARMOT is able to keep track of the proper construction, usage and freeing of all MPI resources, such as communicators, groups, datatypes, etc. It also checks if requests and other arguments (tags, ranks, etc.) are used correctly. The main functionality has been implemented for the C language binding, whereas the functionality for the Fortran language binding is obtained through a wrapper to the C interface. Special attention has to be paid to the verification of the datatypes because they are one of the major differences between the C and the Fortran language binding.
- MARMOT issues warnings if the application relies on non-portable MPI constructs, and error messages if erroneous calls are made. Although high-quality MPI implementations detect some of these errors themselves, there are many cases where they do not give any warnings. For example, non-portable implementation-specific behaviour is not indicated by the implementation, nor are checks performed that would decrease the performance too much, such as consistency checks. MPI implementations tolerate some errors without warnings or crashing, simply giving wrong results. Without the help of a tool like MARMOT, users or developers may thus have a hard time finding these errors.
- Possible race conditions can be identified by locating the calls that may be sources of race conditions. One example is the use of a receive call with the wildcard MPI_ANY_SOURCE as source argument or the wildcard MPI_ANY_TAG as tag argument. One may argue that one does not need a tool to detect this sort of argument as a simple grep command on the source code would give the same result. However, a search command does neither show the execution flow nor will it be able to detect this argument if the application takes functions from some other library with hidden MPI calls. Another source of race conditions is the use of random numbers.
- The tool also contains a mechanism to automatically detect deadlocks and notify the user where and why these have occurred. In general, deadlocks are caused by the non-occurrence of something else, for example mismatched send/receive operations or mismatched collective calls. Currently the deadlock detection is based on a timeout mechanism. MARMOT's debug server surveys the time each process is waiting in an MPI call. If this time exceeds a certain user-defined limit on all processes at the same time, the debug process issues a deadlock warning and allows the user to trace back the last few calls on each node.
- MARMOT supports the complete MPI-1.2 standard, although not all possible tests (such as consistency checks) have been implemented so far. It can be used with any standard-conforming MPI implementation and may thus be deployed on any development platform available to the programmer

1.6. PRODUCT USE CASES

Users of MARMOT just have to relink their application with the MARMOT libraries and run the application with one additional process.

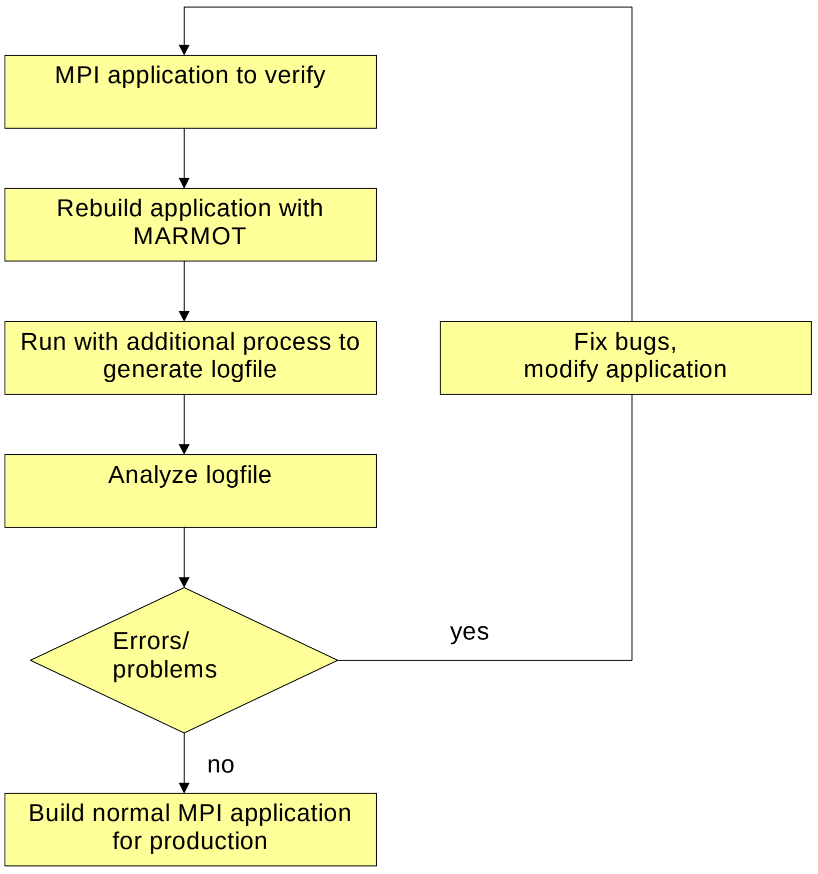
MPI application to verify
Rebuild application with MARMOT
Run with additional process to generate logfile
Analyze logfile
Errors/ problems
yes
no
Fix bugs, modify application
Build normal MPI application for production

figure 2 How to use MARMOT

1.7. PRODUCT COMPONENT MODEL

1.7.1. Debug Server

When an MPI routine is called on an application process, the debug server is informed about the details of the communication (e.g. sender, receiver, message-length, datatype, tag, communicator). The necessary communication between the debug server and the other processes is done with MPI. The debug server has then the possibility to compare the input of all processes. Therefore, the debug server can execute the following tasks:

- Control the execution flow: it is possible that the server does not only monitor the program flow, but does also control which process starts the next MPI call. Since this imposes a huge performance penalty, this will be a user option.
- Signal conditions, e.g. dead-locks.
- Check matching send/receive pairs for consistency: within the MPI standard a message matches if the source, tag and communicator match. It is the responsibility of the user to provide a corresponding datatype on the sender and receiver. The tool will check whether this is the case.
- Output log (report errors, etc.).

1.7.2. Debug Client

The debug client is the component that has the task to check for local consistency on each MPI process.

Examples of Client Checks

- Verification of MPI_Request usage.
- Invalid recycling of active request.
- Invalid use of unregistered request.
- Warning if number of requests is zero.
- Warning if all requests are MPI_REQUEST_NULL.
- Verification of tag range.
- Verification if requested cartesian communicator has correct size.
- Verification of communicator in cartesian calls.
- Verification of groups in group calls.
- Verification of sizes in calls that create groups or communicators.
- Verification if ranges are valid (e.g. in group constructor calls).
- Verification if ranges are distinct (e.g. MPI_Group_incl, -excl).
- Check for pending messages and active requests in MPI_Finalize.
- A detailed enumeration of the checks may be found below.

1.8. DETAILED IMPLEMENTATION MODEL

1.8.1. Errors detected by MARMOT

The file mpo-enums.h contains the enumeration we use for classifying the errors/warnings/notes issued by MARMOT. The enumeration values describe the errors/warnings/notes Marmot checks and are used as return codes for the check methods, for example, checkBlocklength() may return `MARMOT_SUCCESS`, `MARMOT_ERR_BLOCKLENGTH_NEG` or `MARMOT_WARN_BLOCKLENGTH_ZERO`. The nature of the error should be self-evident by the enumeration value, i.e. it should be clear without any further explanation that `MARMOT_ERR_BLOCKLENGTH_NEG` indicates that it is an error to use a negative blocklenth here whereas `MARMOT_WARN_BLOCKLENGTH_ZERO` indicates that using a negative blocklength in this context is not prohibited by the standard but may be an error and is thus issued as a warning.

If new check methods are added to MARMOT's functionality, it should be checked if it can be desribed by an existing enum value or if a new enum value should be added.

```
 /** enum for MARMOT's error return codes.
  *  If you add a new error code, choose a descriptive name.
  *  Distinguish between MARMOT_ERR_FOO, MARMOT_WARN_FOO and
  *  MARMOT_NOTE_FOO.
  */
enum MARMOT_ERR_CODES
{
    //=====================
    // MPI-1.2
    // error types in
    // alphabetical order
    //=====================

    MARMOT_SUCCESS = 0,     // successful return code
    MARMOT_ERR_UNDEFINED,   // undefined error
```

```
MARMOT_ERR_MARMOT,      // implementation error in MARMOT

// blocklength
//======================

// blocklength errors
MARMOT_ERR_BLOCKLENGTH_NEG,  // blocklength is negative

// blocklength warnings
MARMOT_WARN_BLOCKLENGTH_ZERO,  // blocklength is 0

// buffer
//======================

// buffer errors
// MARMOT_ERR_BUFFER,            // invalid buffer pointer
// MARMOT_ERR_TRUNCATE,          // message truncated on receive
MARMOT_ERR_BUFFER_ATTACHED,  // already a bufffer attached
MARMOT_ERR_BUFFER_NOT_ATTACHED, //no buffer attached

// color
//======================

// color errors
MARMOT_ERR_COLOR_NEG,        // color is negative

// color warnings
MARMOT_WARN_COLOR_UNDEFINED,  // color is MPI_UNDEFINED

// communicator
//======================

// communicator errors
MARMOT_ERR_COMM_NULL,   // communicator is MPI_COMM_NULL
MARMOT_ERR_COMM_NOT_VALID,  // communicator is not valid
MARMOT_ERR_COMM_NOT_CARTESIAN,  // communicator is not cartesian
MARMOT_ERR_COMM_NOT_GRAPH,  // communicator is not cartesian
MARMOT_ERR_COMM_NOT_INTERCOMM, // communicator is not intercommunicator
MARMOT_ERR_COMM_INTERCOMM,  // communicator is intercommunicator

// communicator warnings
MARMOT_WARN_COMM_NOT_WORLD,    // communicator is not MPI_COMM_WORLD
MARMOT_WARN_COMM_NULL,      // communicator is MPI_COMM_NULL

// coordinate
//======================

// coordinate errors
MARMOT_ERR_COORD_NEG,  // coordinate is negative
MARMOT_ERR_COORD_TOO_BIG,  // coordinate is out of range

// count
//======================

// count errors
MARMOT_ERR_COUNT_NEG,   // count is negative
MARMOT_ERR_COUNT_ZERO,  // count is 0
// MARMOT_ERR_COUNT_ARRAY_NEG,  //

// count warnings
MARMOT_WARN_COUNT_ZERO, // count is 0
MARMOT_WARN_COUNTS_NOT_EQUAL,  // counts are not equal
MARMOT_WARN_COUNT_UNDEFINED,   // count is MPI_UNDEFINED

// datatype
//======================

// datatype errors
MARMOT_ERR_TYPE_NULL,   // datatype is MPI_DATATYPE_NULL
MARMOT_ERR_TYPE_NOT_VALID,  // datatype is not valid
MARMOT_ERR_TYPE_NOT_COMMITTED,  // datatype is created, not committed
MARMOT_ERR_TYPE_C,  // datatype is C datatype
```

```
MARMOT_ERR_TYPE_F,  // datatype is Fortran datatype
MARMOT_ERR_TYPE_REDUCTION_C,  // datatype is for reduction function (C)
MARMOT_ERR_TYPE_REDUCTION_F,  // datatype is for reduction function (F)
MARMOT_ERR_TYPE_OPTIONAL_C,     // datatype is optional (C)
MARMOT_ERR_TYPE_OPTIONAL_F,     // datatype is optional (F)
MARMOT_ERR_TYPE_CONSTR_DERIVED,  // datatype is for constructing
                                 // derived datatypes
MARMOT_ERR_TYPE_LB,              // datatype is MPI_LB
MARMOT_ERR_TYPE_UB,              // datatype is MPI_UB

// datatype warnings
MARMOT_WARN_TYPE_REDUCTION_C,  //datatype is for reduction function (C)
MARMOT_WARN_TYPE_REDUCTION_F,  //datatype is for reduction function (F)
MARMOT_WARN_TYPE_OPTIONAL_C,  // datatype is optional (C)
MARMOT_WARN_TYPE_OPTIONAL_F,  // datatype is optional (F)

// datatype notes
MARMOT_NOTE_TYPE_NULL,  // datatype is MPI_TYPE_NULL
MARMOT_NOTE_TYPE_ALREADY_COMMITTED_C,  // datatype is already
                                       // committed (C)
MARMOT_NOTE_TYPE_ALREADY_COMMITTED_C_AND_F,  // datatype is already
                                             // committed (C and F)
MARMOT_NOTE_TYPE_ALREADY_COMMITTED_F,  // datatype is already
                                       // committed (F)

// dimension
//======================

// dimension errors
MARMOT_ERR_DIM_NEG,     // dimension is negative
MARMOT_ERR_DIM_TOO_BIG, // dimension is too big
MARMOT_ERR_NODES_NEQ_PRODUCT_DIMS, // nodes are not a multiple of dims

 // dimension warnings
MARMOT_WARN_DIM_ZERO,   // dimension is zero

// displacement
//======================

// displacement errors
MARMOT_ERR_DISPLACEMENT_OVERRIDES_DATA, // displacement overrides data

// displacement notes
MARMOT_NOTE_DISPLACEMENT_READS_DATA, // displacement reads  data /
                                     // holes read from other nodes
MARMOT_NOTE_DISPLACEMENT_OVERRIDES_DATA, // displacement overrides data
MARMOT_NOTE_DISPLACEMENT_DATA_SENT_TWICE,  // displacement sends data
                                           // twice
// edge
//======================

 // edge errors
MARMOT_ERR_EDGE_INVALID,     // edge is invalid
MARMOT_ERR_EDGE_NEG,         // edge is negative

// errhandler
//======================

// errhandler errors
MARMOT_ERR_ERRHANDLER_NOT_VALID,  // errhandler is not valid

// errorcode
//======================

// errorcode errors
MARMOT_ERR_ERRORCODE_NOT_VALID,  // errorcode is not valid

// group
//======================

// group errors
// MARMOT_ERR_GROUP_NULL,             // group is MPI_GROUP_NULL
```

```
MARMOT_ERR_GROUP_NOT_VALID,        // group is not valid

// group warnings
MARMOT_WARN_GROUP_NULL,             // group is MPI_GROUP_NULL

// index
//======================

// index errors
MARMOT_ERR_INDEX_NEG,             // index is negative

// index warnings
MARMOT_WARN_INDEX_LT_SIZE,  // index is less than size of communicator

// leader
//======================

// leader errors
// Should we also distinguish between errors for remote
// and for local leaders?
MARMOT_ERR_LEADER_ANY_SOURCE,       // leader is MPI_ANY_SOURCE
MARMOT_ERR_LEADER_NEG,              // leader is negative
MARMOT_ERR_LEADER_OUT_OF_COMM,      // leader is out of communicator

// neighbor
//======================

// neighbor errors
MARMOT_ERR_NEIGHBORS_NEG,          // number of neighbors is negative

// operator
//======================

// operator errors
MARMOT_ERR_OP_NULL,             // operator is MPO_OP_NULL
MARMOT_ERR_OP_NOT_VALID,        // operator is not valid

// pending messages
//======================

// pending messages warnings
MARMOT_WARN_PENDING_MESSAGES_LEFT, // pending messages are left

// range
//======================

// range errors
MARMOT_ERR_RANK_FIRST_NEG,  // first rank is negative
MARMOT_ERR_RANK_LAST_NEG,   // last rank is negative
MARMOT_ERR_RANK_FIRST_TOO_BIG,  // first rank is too big
MARMOT_ERR_RANK_LAST_TOO_BIG,   // last rank is too big
MARMOT_ERR_RANK_COMPUTED_NEG, // computed rank is negative
MARMOT_ERR_RANK_COMPUTED_TOO_BIG, // computed rank is too big
MARMOT_ERR_RANK_COMPUTED_TWICE,  // rank is computed twice
MARMOT_ERR_STRIDE_NEG,  // stride is negative
MARMOT_ERR_STRIDE_ZERO, // stride is zero
MARMOT_ERR_STRIDE_POS,  // stride is positive

// rank
//======================

// rank errors (see also range errors)
// MARMOT_ERR_RANK,               // invalid rank
MARMOT_ERR_RANK_NEG,            // rank is negativ
MARMOT_ERR_RANK_PROC_NULL,      // rank is MPI_PROC_NULL
MARMOT_ERR_RANK_ANY_SOURCE,     // rank is MPI_ANY_SOURCE
MARMOT_ERR_RANK_TOO_BIG,        // rank is too big
// MARMOT_ERR_DUP_RANK,
// MARMOT_ERR_RANK_ARRAY,
// MARMOT_ERR_LOCAL_RANK,
// MARMOT_ERR_REMOTE_RANK,
// rank warnings
MARMOT_WARN_RANK_ANY_SOURCE,    // rank is MPI_ANY_SOURCE

// rank notes
MARMOT_NOTE_RANK_PROC_NULL,     // rank is MPI_PROC_NULL

// request
//======================

// request errors
MARMOT_ERR_REQUEST_COUNT_NEG,   // count of requests is negative
MARMOT_ERR_REQUEST_NOT_VALID,   // request is not valid
MARMOT_ERR_REQUEST_STILL_USED,  // request is still in use
MARMOT_ERR_REQUEST_NOT_PERSISTENT,  // request is not persistent
MARMOT_ERR_REQUEST_OUTCOUNT_UNDEFINED,  // outcount is MPI_UNDEFINED

// request warnings
MARMOT_WARN_REQUEST_COUNT_ZERO, // count of requests is zero
MARMOT_WARN_REQUEST_NULL,       // request is MPI_REQUEST_NULL
MARMOT_WARN_REQUEST_ACTIVE_NOT_PERSISTENT, // request is not persistent
                                           // and active
MARMOT_WARN_REQUEST_ACTIVE_PERSISTENT,  // request is persistent and
                                        // still active
MARMOT_WARN_REQUEST_INACTIVE_PERSISTENT,  // request is persistent
                                          // and inactive
MARMOT_WARN_REQUEST_OUTCOUNT_UNDEFINED,  // outcount is MPI_UNDEFINED
MARMOT_WARN_REQUESTS_LEFT,               // requests are left

// size
//======================

// size errors
MARMOT_ERR_SIZE_NEG,             // size is negative

// size warnings
MARMOT_WARN_SIZE_ZERO,             // size is zero

// tag
//======================

// tag errors
MARMOT_ERR_TAG_NEG,             // tag is negative
MARMOT_ERR_TAG_TOO_BIG,         // tag is too big ( > TAG_UB)
MARMOT_ERR_TAG_ANY_TAG,         // tag is MPI_ANY_TAG

// tag warnings
MARMOT_WARN_TAG_TOO_BIG,        // tag is too big ( > guaranteed range,
                                //  but < TAG_UB)

// topology
//======================

// topology errors (see also dimension errors, coordinates errors)
MARMOT_ERR_TOPOLOGY_NEW_GT_COMM_OLD, // new topology greater than
                                     // old communicator
MARMOT_ERR_TOPOLOGY_CART_NEW_GT_COMM_OLD, // new cartesian topology
                                          // greater than
                                          // old communicator


// topology warnings (see also dimension errors, coordinates errors)
MARMOT_WARN_TOPOLOGY_NEW_LT_COMM_OLD, // new topology smaller than
                                      // old communicator
MARMOT_WARN_TOPOLOGY_CART_NEW_LT_COMM_OLD, // new cartesian topology
                                           // smaller than
                                           // old communicator

// misc
//======================

// misc
// MARMOT_ERR_PENDING,           // Pending request
```

```
//======================
// MPI-2
// file I/O
// not implemented yet
// error types in
// alphabetical order
//======================

// amode
//======================

// amode errors

// file access mode is not the same for all processes
MARMOT_ERR_AMODE_DIFF,

// MPI_MODE_RDONLY is specified
MARMOT_ERR_AMODE_RDONLY,

// MPI_MODE_WRONLY is specified
MARMOT_ERR_AMODE_WRONLY,

// exactly one of MPI_MODE_RDONLY, MPI_MODE_RDWR, MPI_MODE_WRONLY
// must be specified
// MARMOT_ERR_AMODE_RDONLY_AND_RDWR,
// MARMOT_ERR_AMODE_RDONLY_AND_WRONLY,
// MARMOT_ERR_AMODE_RDWR_AND_WRONLY,
MARMOT_ERR_AMODE_RDONLY_RDWR_WRONLY_NOT_SET_EXACTLY_ONE,

// MPI_MODE_RDONLY is specified with MPI_MODE_CREATE
MARMOT_ERR_AMODE_RDONLY_AND_CREATE,

// MPI_MODE_RDONLY is specified with MPI_MODE_EXCL
MARMOT_ERR_AMODE_RDONLY_AND_EXCL,

// MPI_MODE_RDWR is specified with MPI_MODE_SEQUENTIAL
MARMOT_ERR_AMODE_RDWR_AND_SEQUENTIAL,

// MPI_MODE_UNIQUE_OPEN is specified
MARMOT_ERR_AMODE_UNIQUE_OPEN,

// MPI_MODE_SEQUENTIAL is specified
MARMOT_ERR_AMODE_SEQUENTIAL,

// MPI_MODE_SEQUENTIAL is specified, but not MPI_DISPLACEMENT_CURRENT
MARMOT_ERR_AMODE_SEQUENTIAL_AND_NOT_DISPLACEMENT_CURRENT,

// MPI_MODE_SEQUENTIAL is not specified, but MPI_DISPLACEMENT_CURRENT
MARMOT_ERR_AMODE_SEQUENTIAL_NOT_SET_BUT_DISPLACEMENT_CURRENT,

// amode notes

// amode is MPI_DELETE_ON_CLOSE
MARMOT_NOTE_AMODE_DELETE_ON_CLOSE,

// communicator - see also above
//======================

// communicator errors

// communicator is intercommunicator, see above
// MARMOT_ERR_COMM_INTERCOMM,

// data representation
//======================

// data representation errors

// data representation is not the same for all processes
MARMOT_ERR_DATAREP_DIFF,
// data representation string is larger than MPI_MAX_DATAREP_STRING
MARMOT_ERR_DATAREP_GT_MAX_DATAREP_STRING,

// data representation warnings

// data representation is not large enough
MARMOT_WARN_DATAREP_TOO_SMALL,

// displacement
//======================

// displacement errors

// displacements in typemap of filetype are negative
MARMOT_ERR_FILETYPE_DISPLACEMENTS_NEG,

// displacements in typemap of filetype are decreasing
MARMOT_ERR_FILETYPE_DISPLACEMENTS_DECREASE,

// displacement warnings

// values of data in new regions (those locations with displacements
// between old file size and size) are undefined
MARMOT_WARN_DATA_IN_NEW_REGION_UNDEFINED,


// datatype - see also above datatype errors
//======================

// datatype errors

// datatype has overlapping regions
MARMOT_ERR_TYPE_OVERLAPPING,

// etype - see also above datatype errors
//======================

// etype errors

// extent of etype is not the same on all processes
MARMOT_ERR_ETYPE_EXTENT_DIFF,

// etype is not valid, see above data type errors
MARMOT_ERR_ETYPE_NOT_VALID,

// etype has overlapping regions
MARMOT_ERR_ETYPE_OVERLAPPING,

// filetype - see also above datatype errors
//======================

// filetype errors

// extent of hole in the filetype is not a multiple of the extent
// of the etype
MARMOT_ERR_FILETYPE_EXTENT_HOLE_NO_MULTIPLE_EXTENT_ETYPE,

// filetype is not valid, see above data type errors
MARMOT_ERR_FILETYPE_NOT_VALID,

// filetype has overlapping regions
MARMOT_ERR_FILETYPE_OVERLAPPING,

// filetype notes

// filetype has holes
MARMOT_NOTE_FILETYPE_HAS_HOLES,

// filename
//======================

// filename errors
```

```
// filenames do not reference the same file
MARMOT_ERR_FILE_DIFF,

// file does not exist
MARMOT_ERR_FILE_DOES_NOT_EXIST,

// filename warnings

// file still in use
MARMOT_WARN_FILE_IN_USE,

// group - see above
//======================

// group errors
// MARMOT_ERR_GROUP_NOT_VALID,        // group is not valid

// group warnings
// MARMOT_WARN_GROUP_NULL,             // group is MPI_GROUP_NULL

// info
//======================

// info errors

// info entry is not the same for all processes
MARMOT_ERR_INFO_DIFF,

// info notes

// implementation is free to ignore info
MARMOT_NOTE_INFO_IGNORED,

// offset
//======================

// offset errors

// offset is negative
MARMOT_ERR_OFFSET_NEG,

// offset reaches a negative position in the view
MARMOT_ERR_OFFSET_TO_NEG_POS_IN_VIEW,

// size
//======================

// size errors

// size is not the same for all processes
MARMOT_ERR_FILE_SIZE_DIFF,

// size is negative
MARMOT_ERR_FILE_SIZE_NEG,

// size notes
MARMOT_NOTE_FILE_SIZE_LT_OR_EQ_CURRENT_SIZE,

// misc
//======================

// misc errors

// request left on file handle, see above
MARMOT_ERR_REQUESTS_NONBLOCKING_LEFT,

// split collective operation left on file handle
MARMOT_ERR_OPERATION_SPLIT_COLLECTIVE_LEFT,

// more than one active split collective operation on file handle
MARMOT_ERR_OPERATION_SPLIT_COLLECTIVE_COUNT_GT_ONE,
        // no matching begin call
        MARMOT_ERR_NO_MATCHING_BEGIN,

        // misc warnings

        // open file left (after MPI_Finalize)
        MARMOT_WARN_FILE_OPEN_LEFT,

        // request left on file handle, see above
        MARMOT_WARN_REQUESTS_NONBLOCKING_LEFT,

        // split collective operation left on file handle
        MARMOT_WARN_OPERATION_SPLIT_COLLECTIVE_LEFT,

        // whence argument is not the same on all processes
        MARMOT_ERR_WHENCE_DIFF,

        // flag argument is not the same on all processes
        MARMOT_ERR_FLAG_DIFF,

        // file handle argument is not the same on all processes
        MARMOT_ERR_FILEHANDLE_DIFF,

        MARMOT_LAST_ERR_CODE
};                          // end enum MARMOT_ERR_CODES
```

1.8.2. DebugServer Class

- Purpose: The DebugServer class represents the server for debugging, with the following tasks:
    - Control the execution flow.
    - Detect dead-locks.
    - Check consistency of matching send/receive pairs.
    - Output log.
- Public Methods:
    - DebugServer (Messenger &communication_channel)

      sets up server with provided communication_channel.
    - void runMainLoop ()

      The loop will never return, but exit from the application on request.

1.8.3. Space, Object and Derived Classes

- The MPO_Space class is the base class at the top of the class hierarchy, see the figure 3 and the description below. This class does not contain any public methods, but protected methods and attributes building the bridge between MPI datatypes, communicators, groups, operators etc. and our MPO datatypes, communicators, etc. For example, all predefined communicators are automatically inserted in the set our_comms. All newly created communicators are inserted when created and removed when freed. So we can check if a used communicator is valid or not.
- The MPO_Object class is derived from the MPO_Space class and contains the three subclasses MPO_Callobject, MPO_Typeobject and MPO_State, see below.
    - All the 129 MPI-1.2 calls have been implemented. All MPI calls are represented as classes derived from the MPO_Callobject class.
        - The hierarchy of the class MPO_Callobject consists of several derived classes that gather classes representing similar MPI calls, e.g. the class

MPO_Collective is derived from the class MPO_Callobject and contains all classes representing MPI calls for collective communication (MPI_Bcast etc.). Thus having 129 classes representing MPI calls plus several base classes, only the base classes are described in detail below, whereas the MPI call classes are mentioned by name only. Information on the MPI calls themselves, on the parameters they take and on the errors that should be prevented can be found in the MPI standards. For example, when using MPI_Bcast the validity of the count, datatype, root and communicator parameters should be checked.

- MPO_Callobject defines the interface and the protocol used in the library between client and debug server.
- The client checks described in section 4 are achieved by implementing adequate member functions, e.g. checkPrerequisites(), checkTag(), checkOp(), checkRoot(), checkRank() …
- Advantages:
  - The server only handles MPO_Calls (no change necessary if new calls are implemented).
  - Definition of protocol and behaviour of the client in one place.
  - No need to handle function pointers.

  - The MPO_Typeobject class represents MPI types (communicators, datatypes, groups and operators). These four types correspond to the classes MPO_Comm, MPO_Datatype, MPO_Group and MPO_Op, which are derived from the base class MPO_Typeobject.
  - The MPO_State class represents the state of an MPI process, i.e. whether it is pending, finished etc.

- In general, classes are forced to implement the following methods:
  - getFamilyId() and getId()

    are to provide a unique identity (a unique enumeration value) for each family of objects respectively for each object.
  - save() and restore()

    are responsible for serializing and de-serializing an object to ostream and from

    istream.

  Additionally, classes representing MPI calls have the following methods:
  - Member functions for client checks, e.g. checkPrerequisites() etc.
  - getCallName()

    gets the name of the MPI call.
  - callNativeMPI()

    does the MPI call by using client check functions.
- We need to map MPI communicators to new communicators excluding the debug server. This mapping can be guaranteed by introducing new types:
  - int MPI_Barrier(MPI_Comm comm);
  - int MPO_Barrier(MPO_Comm a_comm){

```
        return PMPI_Barrier(a_comm);
    }
```

Mechanism:

- The mapping is done in the constructor of the MPO_Comm class.
- A type conversion operator enables that an object of type MPO_Comm can be used wherever a variable of type MPI_Comm is expected.
- The declaration of MPO_Barrier guarantees that no invalid communicators are used inside the profiling interface/debugger.

1.8.3.1. Class Space

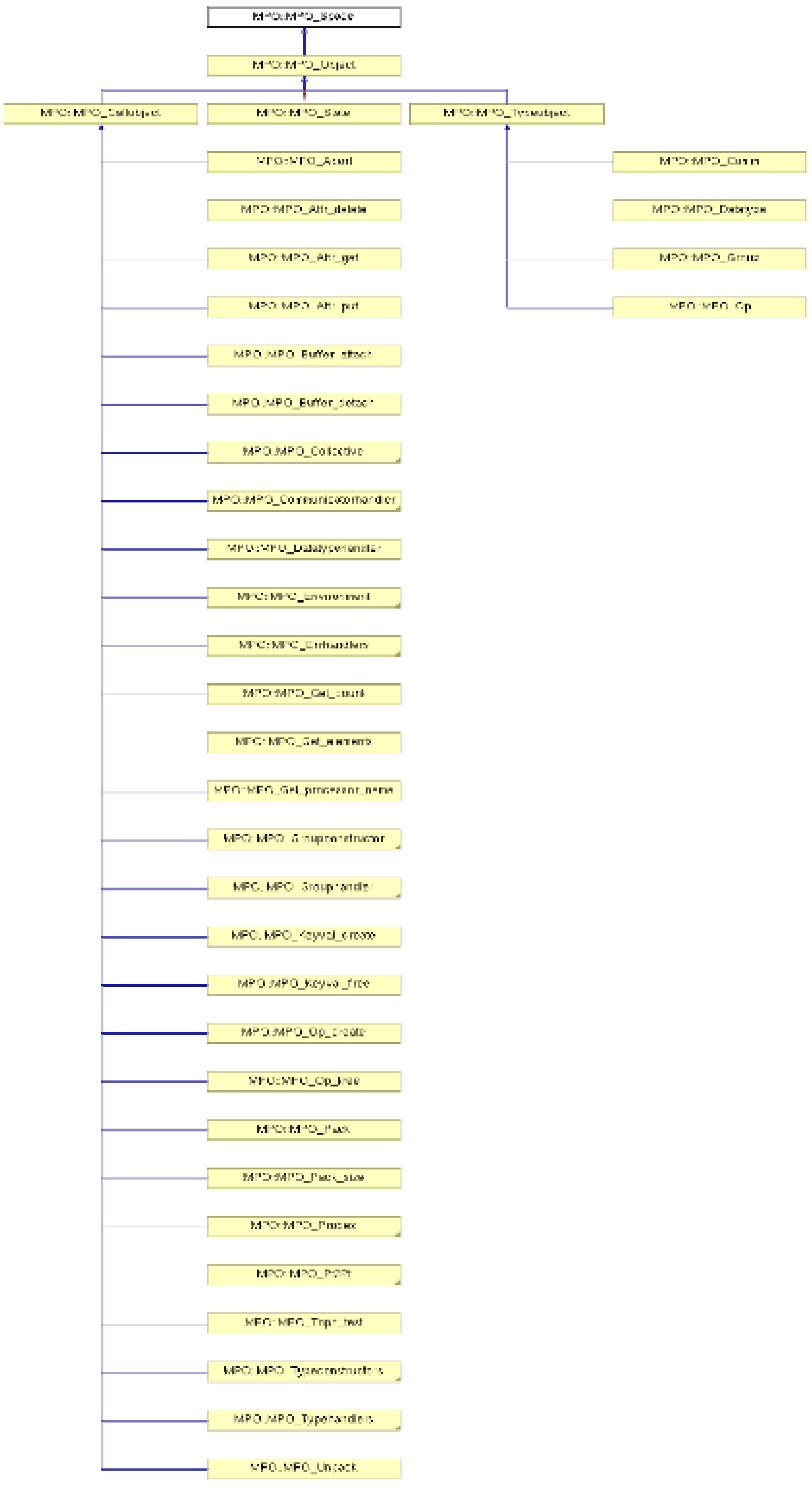

figure 3 Class Space

- Purpose: MPO_Space provides sets to handle communicators, groups and operators and maps to handle requests, datatypes and errhandlers.
- Public Methods: none.

1.8.3.2. Class Object

- Purpose: MPO_Object represents calls, types and messages. All MPO_Objects should be transferable to and from a stream. The member functions getId(), getFamilyId(), save() and restore() are intended to provide this functionality. A general description of these functions is given above in section 7.1.2.
- Public Methods:
  - virtual int getId () const=0
  - virtual int getFamilyId () const=0
  - virtual std::ostream & save (std::ostream &out) const=0
  - virtual std::istream & restore (std::istream &in)=0

The diagram above shows that the following three classes are derived from the class MPO_Object.

1.8.3.2.1. Class State

- Purpose: This class represents the state of an MPI process. These states are given by the enumeration values MPO_STARTED, MPO_CALLED, MPO_PENDING, MPO_FINISHED, APPL_RUNNING and LAST_STATE_ID.
- Public Methods:
  - MPO_State (MPO_State_id state)

    uses the states enumerated above.
  - virtual int getId () const
  - virtual int getFamilyId () const
  - virtual std::ostream & save (std::ostream &out) const
  - virtual std::istream & restore (std::istream &in)

1.8.3.2.2. Class Typeobject

- Purpose: MPO_Typeobject represents MPI_Types (communicators, datatypes, groups and operators). For each MPI_Type MPI_foo there is one object MPO_foo. The constructor takes an argument of type MPI_foo. These arguments are stored in private variables. Necessary conversions take place in the constructor. For ease of use a conversion function to MPI_foo exists. Therefore an object of type MPO_foo can be used, wherever an object of type MPI_foo is required. Vice versa, an argument of type MPI_foo can be passed, wherever type MPO_foo is expected.
- Public Methods:
  - virtual int getFamilyId () const

1.8.3.2.2.1. Class Comm

- Purpose: The class MPO_Comm represents the type MPI_Comm.
- Public Methods: Apart from the usual member functions (constructors, destructor, getId(), save() and restore(), see above), we have additionally:
  - operator MPI_Comm () const

    returns the corresponding MPO communicator.

1.8.3.2.2.2. Class Datatype

- Purpose: The class MPO_Datatypes represents the type MPI_Datatype.
- Public Methods: Apart from the usual member functions (constructors, destructor, getId(), save() and restore(), see above), we have additionally:
  - operator MPI_Datatype () const

    returns the corresponding MPO datatype.

1.8.3.2.2.3. Class Group

- Purpose: The class MPO_Group represents the type MPI_Group.
- Public Methods: Apart from the usual member functions (constructors, destructor, getId(), save() and restore(), see above), we have additionally:
  - operator MPI_Group () const

    returns the corresponding MPO group.

1.8.3.2.2.4. Class Op

- Purpose: The class MPO_Op represents the type MPI_Op.
- Public Methods: Apart from the usual member functions (constructors, destructor, getId(), save() and restore(), see above), we have additionally:
  - operator MPI_Op () const

    returns the corresponding MPO operator.

1.8.3.2.3. Class Callobject

- Purpose: MPO_Callobject represents MPI_Calls. Several classes are derived from this class, see below. For each MPI call MPI_foo there is one object MPO_foo.
  - The constructor takes all arguments of the MPI call. These arguments are stored in private variables.
  - The call to the MPI_foo is made when the member function callNativeMPI() is called.
- Public Methods:
  - virtual int getFamilyId () const
  - virtual int callNativeMPI ()=0

    does the MPI-call.
  - virtual int callDebugServer ()

    handles communication with the debug process.
  - virtual string getCallName () const=0

is just to get the name of the MPI call.

1.8.3.2.3.1. Class Collective

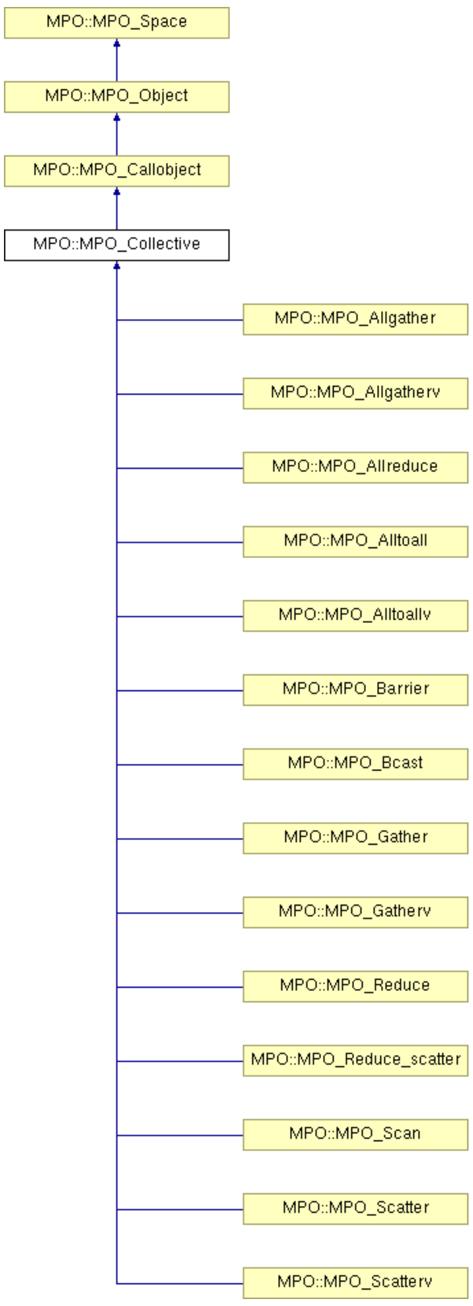

figure 4 Class Collective

- Purpose: All calls concerning collective communication are derived from this class. The check of the parameters is implemented as several member functions.
- Public Methods: Apart from the usual member functions (constructors, destructor, getFamilyId(), callNativeMPI(), getCallName(), see above), we have additional member functions:
    - virtual int checkPrerequisites ()

checks whether the communicator is valid.

- virtual int checkCountType (int count, MPO_Datatype datatype, int kind, int red) checks whether count and datatype are valid. Kind stands for receive-/ sendcount, red is set true for MPI_Reduce or MPI_Allreduce.
- virtual int checkRoot (int root)

  checks whether root is a valid rank.
- virtual int checkOp (MPO_Op op)

  checks if op is a valid operator.

1.8.3.2.3.2. Class Communicatorhandler

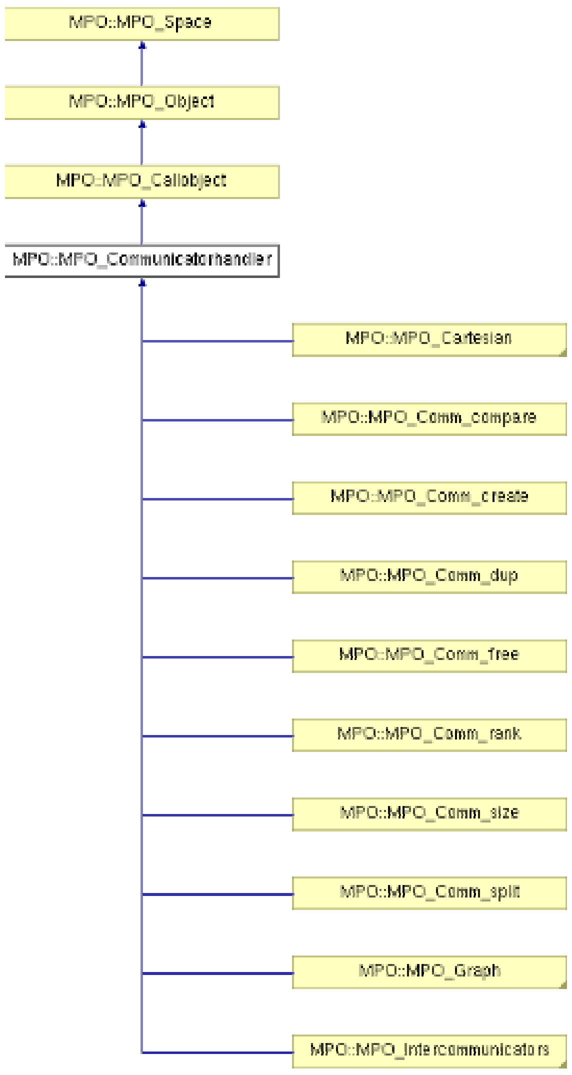

figure 5 Class Communicatorhandler

- Purpose: All calls for communicator-handling are derived from this class.
- Public Methods: Apart from the usual member functions (constructors, destructor, getFamilyId(), callNativeMPI(), getCallName(), see above), we have additionally:

- o virtual int checkPrerequisites ()

  checks whether communicator is valid.

1.8.3.2.3.2.1. Class Cartesian

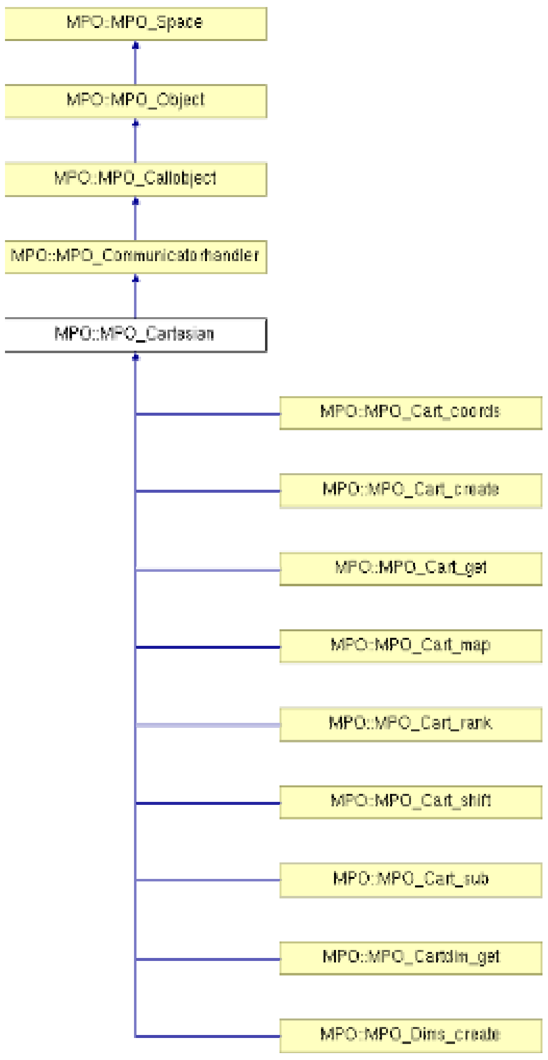

figure 6 Class Cartesian

- Purpose: All calls concerning cartesian coordinates are derived from this class.
- Public Methods: Apart from the usual member functions (constructors, destructor, getFamilyId(), callNativeMPI(), getCallName(), see above), we have additionally:
  - o virtual int checkPrerequisites ()

    checks if communicator is a valid cartesian one.

1.8.3.2.3.2.2. Class Graph

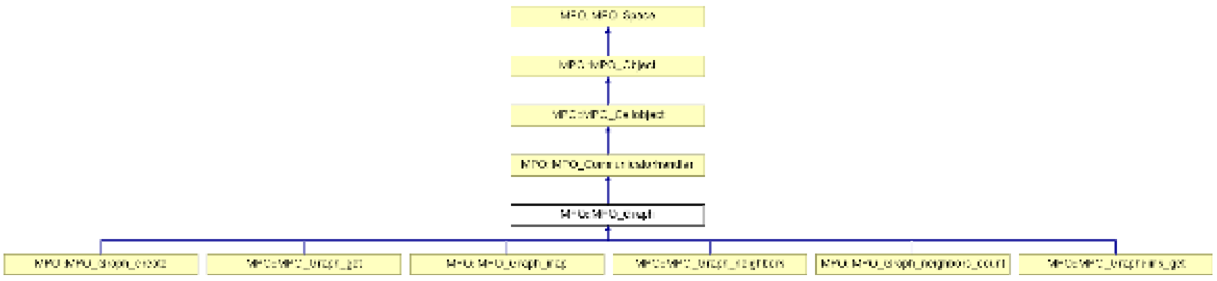

figure 7 Class Graph

- Purpose: All calls concerning graph coordinates are derived from this class.
- Public Methods: Apart from the usual member functions (constructors, destructor, getFamilyId(), callNativeMPI(), getCallName(), see above), we have additionally:
    - o virtual int checkPrerequisites ()

      checks whether communicator is valid for graph operation.

1.8.3.2.3.2.3. Class Intercommunicators

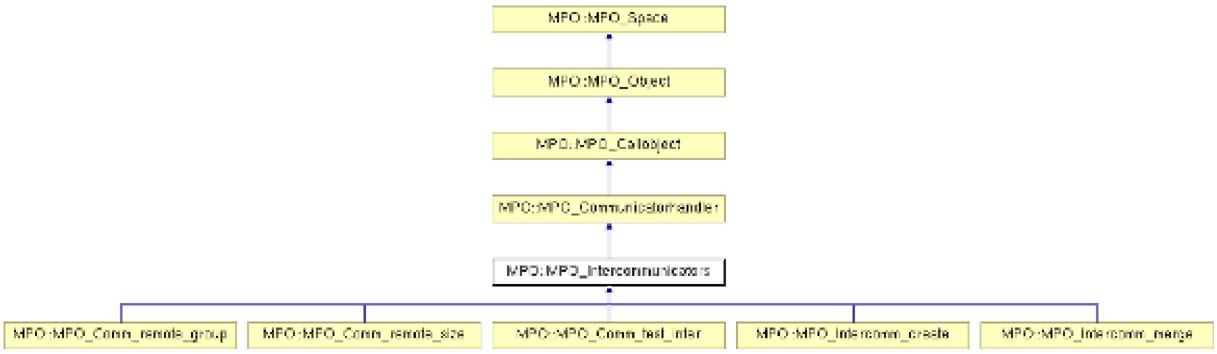
MPO::MPO_Space
MPO::MPO_Object
MPO::MPO_Callobject
MPO::MPO_Communicatorhandler
MPO::MPO_Intercommunicators
MPO::MPO_Comm_remote_group
MPO::MPO_Comm_remote_size
MPO::MPO_Comm_test_inter
MPO::MPO_Intercomm_create
MPO::MPO_Intercomm_merge

figure 8 Class Intercommunicators

- Purpose: All calls handling inter-communicators are derived from this class.
- Public Methods: Apart from the usual member functions (constructors, destructor, familyId(), callNativeMPI(), getCallName(), see above), we have additionally:
    - o virtual int checkPrerequisites ()

      checks whether communicator is valid.

1.8.3.2.3.3. Class Environment

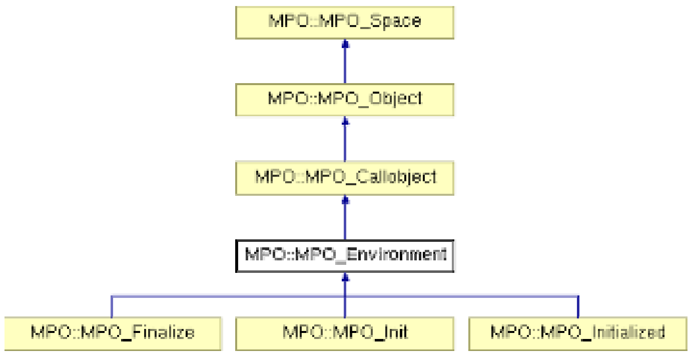

figure 9 Class Environment

- Purpose: This class represents calls concerning MPI itself.
- Public Methods: getFamilyId(), callNativeMPI() and getCallName(), see above.

1.8.3.2.3.4. Class Errhandlers

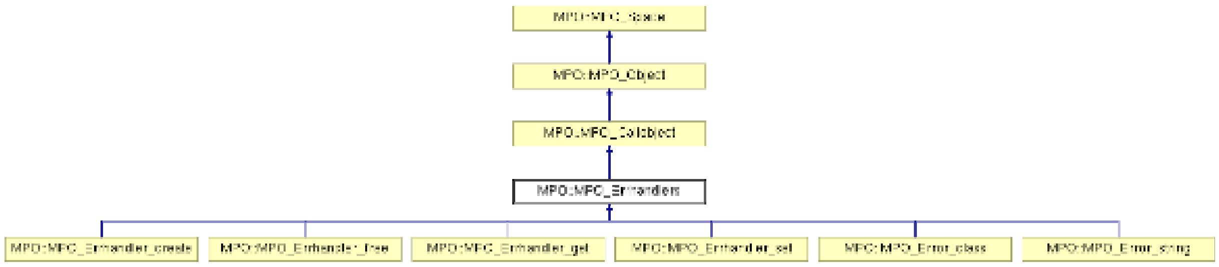

figure 10 Class Errhandlers

- Purpose: All calls concerning error-handling are derived from this class.
- Public Methods: familyId(), callNativeMPI() and getCallName(), see above.

1.8.3.2.3.5. Class Groupconstructor

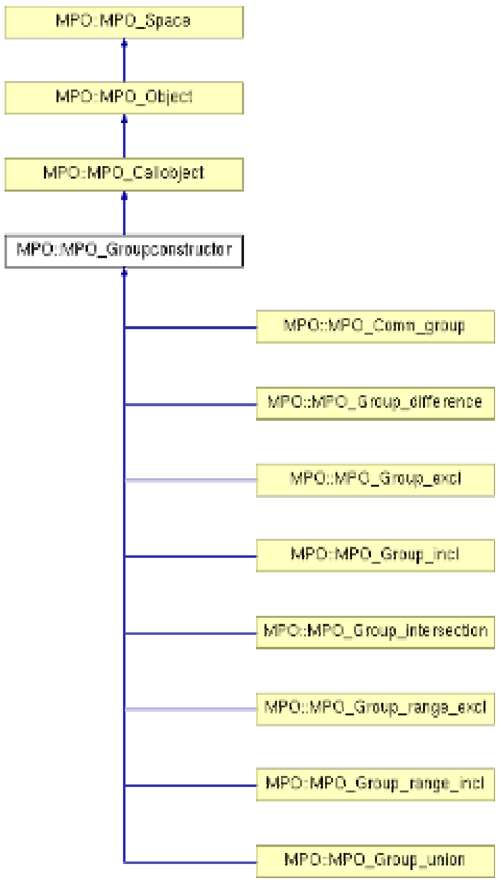
MPO::MPO_Space
MPO::MPO_Object
MPO::MPO_Callobject
MPO::MPO_Groupconstructor
MPO::MPO_Comm_group
MPO::MPO_Group_difference
MPO::MPO_Group_excl
MPO::MPO_Group_incl
MPO::MPO_Group_intersection
MPO::MPO_Group_range_excl
MPO::MPO_Group_range_incl
MPO::MPO_Group_union

figure 11 Class Groupconstructor

- Purpose: All calls constructing new groups are derived from this class.
- Public Methods: Apart from the usual member functions (constructors, destructor, getFamilyId(), callNativeMPI(), getCallName(), see above), we have additionally:
  - virtual int checkPrerequisites ()

    checks whether groups are valid.
  - virtual int insertGroup ()

    adds a new group to our set.

1.8.3.2.3.6. Class Grouphandler

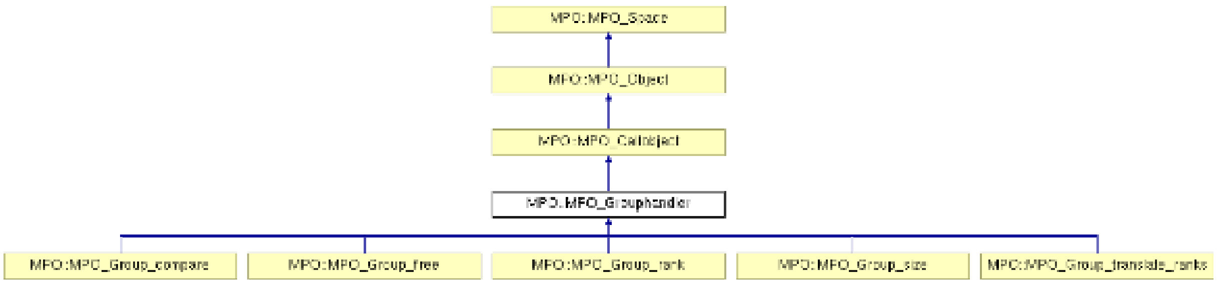

figure 12 Class Grouphandler

- Purpose: All calls handling groups are derived from this class.
- Public Methods: Apart from the usual member functions (constructors, destructor, getFamilyId(), callNativeMPI(), getCallName(), see above), we have additionally:
  - virtual int checkPrerequisites ()

    checks whether groups are valid.

1.8.3.2.3.7. Class Probes

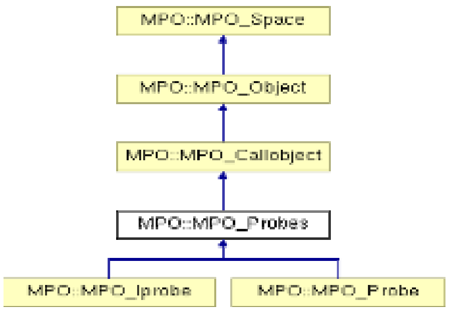

figure 13 Class Probes

- Purpose: MPO_Iprobe and MPO_Probe are derived from this class.
- Public Methods: Apart from the usual member functions (constructors, destructor, getFamilyId(), callNativeMPI(), getCallName(), see above), we have additionally:
  - virtual int checkPrerequisites ()

    checks whether tag and communicator are valid and whether source has valid rank.

1.8.3.2.3.8. Class Pt2Pt

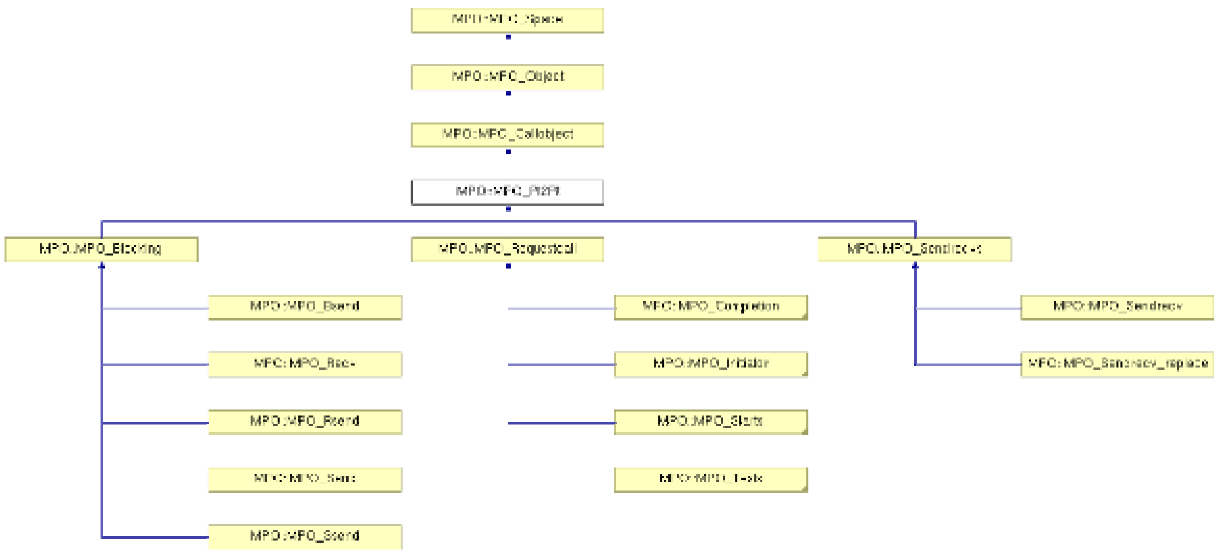

figure 14 Class Pt2Pt

- Purpose: This class represents point-to-point-calls.
- Public Methods: getFamilyId(), callNativeMPI() and getCallName(), see above.

1.8.3.2.3.8.1. Class Blocking

- Purpose: Most of the blocking calls are derived from this class.
- Public Methods: Apart from the usual member functions (constructors, destructor, familyId(), callNativeMPI(), getCallName(), see above), we have additionally:
    - virtual int checkPrerequisites ()

      checks whether tag, rank, datatype and communicator are valid.

1.8.3.2.3.8.2. Class Sendrecvs

- Purpose: Sendrecv and Sendrecv_replace are derived from this class.
- Public Methods: Apart from the usual member functions (constructors, destructor, getFamilyId(), callNativeMPI(), getCallName(), see above), we have additionally:
    - virtual int checkPrerequisites ()

      checks whether tag, rank, datatype, count and communicator are valid.

1.8.3.2.3.8.3. Class Requestcall

- Purpose: All calls using requests are derived from this class.
- Public Methods: getFamilyId(), callNativeMPI() and getCallName(), see above.

1.8.3.2.3.8.3.1. Class Completion

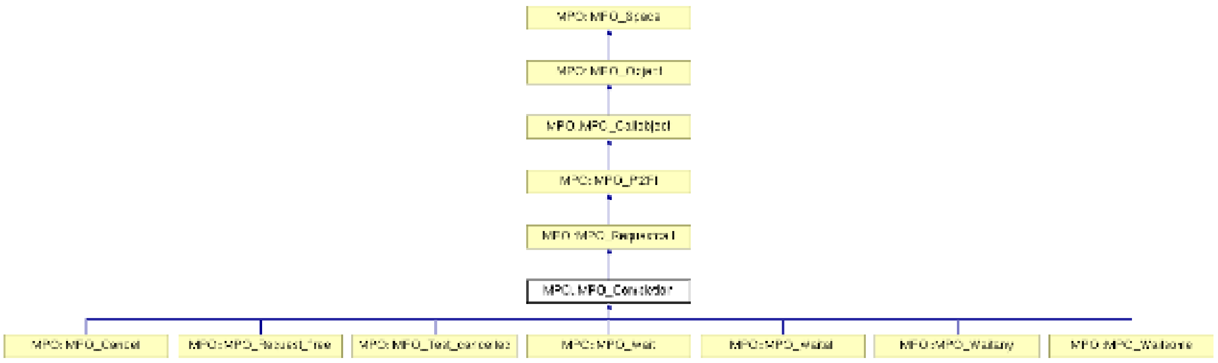

figure 15 Class Completion

- Purpose: This class represents calls that complete communication using requests.
- Public Methods: Apart from the usual member functions (constructors, destructor, getFamilyId(), callNativeMPI(), getCallName(), see above), we have additionally:
  - virtual int checkPrerequisites ()

    checks whether number of requests and requests are valid.
  - virtual int eraseRequest (int *my_outcount, int *my_index_array)

    checks whether request is to remove from set our_requests.

1.8.3.2.3.8.3.2. Class Initiator

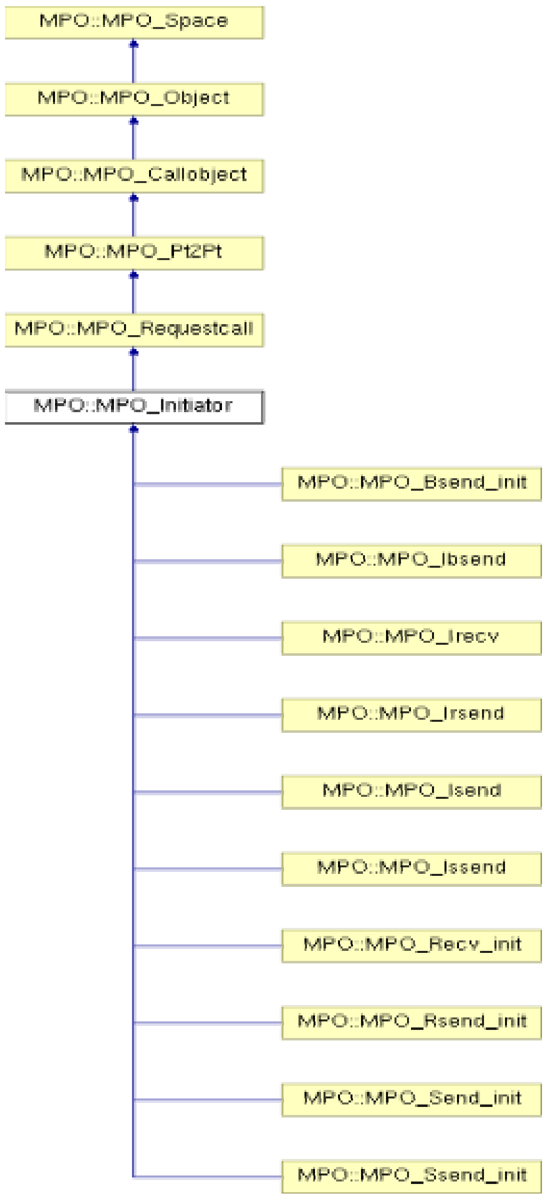

figure 16 Class Initiator

- Purpose: This class represents calls that initiate communication using requests.
- Public Methods: Apart from the usual member functions (constructors, destructor, getFamilyId(), callNativeMPI(), getCallName(), see above), we have additionally:
  - virtual int checkPrerequisites (int id)

    checks correctness of count, datatype, request, rank and tag.

1.8.3.2.3.8.3.3. Class Starts

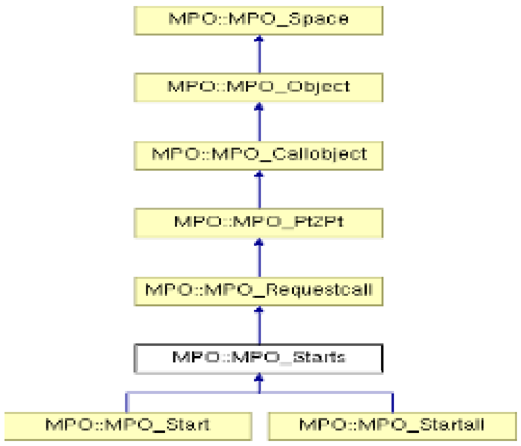

figure 17 Class Starts

- Purpose: This class represents calls that start persistent communication requests.
- Public Methods: Apart from the usual member functions (constructors, destructor, getFamilyId(), callNativeMPI(), getCallName(), see above), we have additionally:
  - o virtual int checkPrerequisites ()

    checks whether requests are valid.

1.8.3.2.3.8.3.4. Class Tests

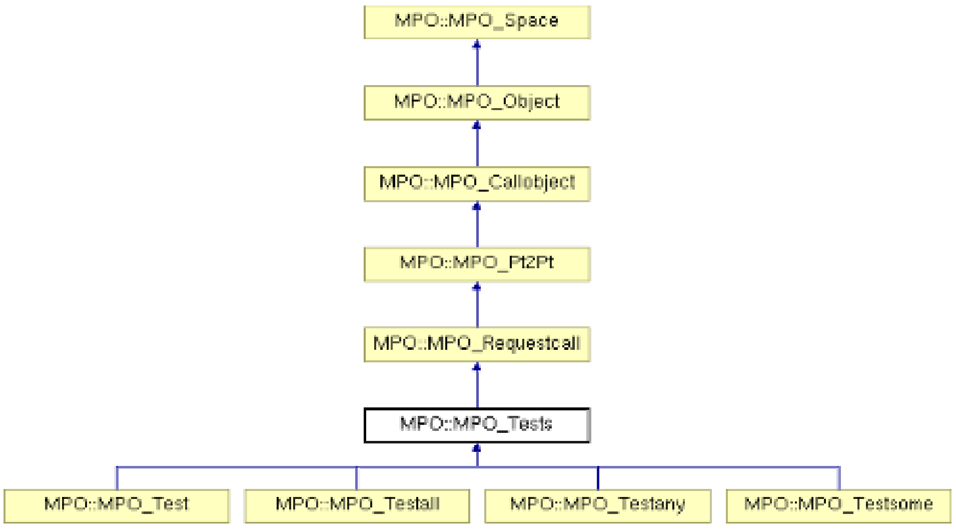

figure 18 Class Tests

- Purpose: This class represents calls that test communication using requests.
- Public Methods: Apart from the usual member functions (constructors, destructor, getFamilyId(), callNativeMPI(), getCallName(), see above), we have additionally:
  - o virtual int checkPrerequisites ()

    checks whether requests are valid.

- virtual int eraseRequest (int *my_outcount, int *my_index_array)
  checks whether request is to remove from set our_requests.

1.8.3.2.3.9. Class Typeconstructors

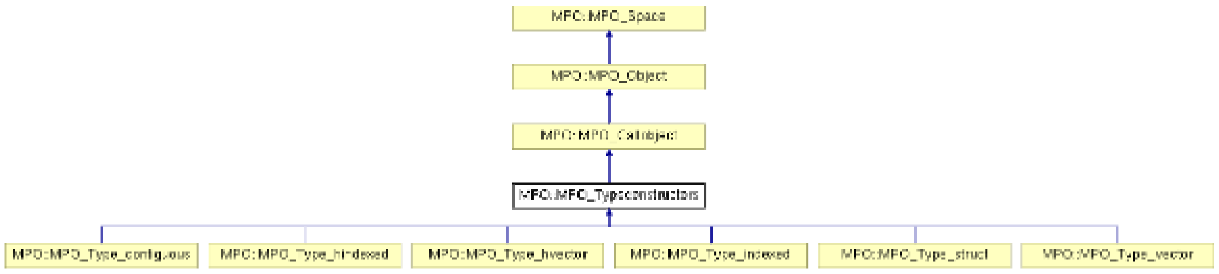

figure 19 Class Typeconstructors

- Purpose: All calls constructing new datatypes are derived from this class.
- Public Methods: Apart from the usual member functions (constructors, destructor, familyId(), callNativeMPI(), getCallName(), see above), we have additionally:
  - virtual int checkPrerequisites ()
    checks whether count and datatype are valid.

1.8.3.2.3.10. Class Typehandlers

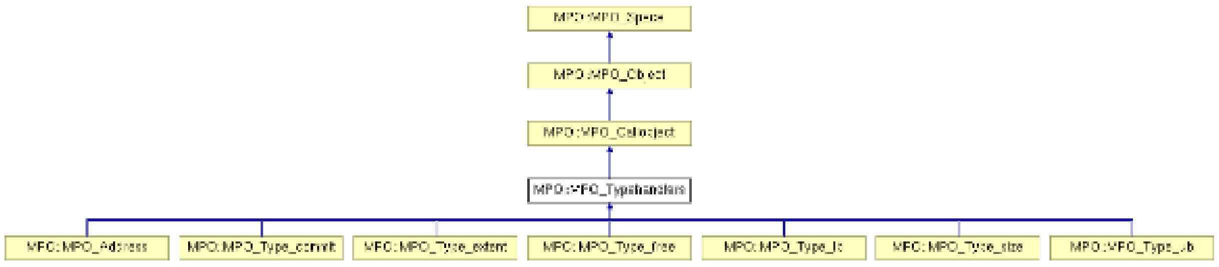

figure 20 Class Typehandlers

- Purpose: All calls handling datatypes are derived from this class.
- Public Methods: Apart from the usual member functions (constructors, destructor, getFamilyId(), callNativeMPI(), getCallName(), see above), we have additionally:
  - virtual int checkPrerequisites ()
    checks whether datatypes are valid.

1.9. HOW TO ADD NEW CALLS AND CLASSES

- TODO: examples
- Check if there already is an appropriate base class, if not, you may want to create one.
- Implement the MPI call according to this example. TODO

- Use javadoc comments for documentation. They serve also as base for the documentation generated by doxygen.

1.10. PRODUCT INTERFACES

The tool provides the API as defined in the MPI standard version 1.2. More precise details follow below.

1.10.1. Output

- Violation of the MPI standard will be reported as error.
- Unusual behaviour or possible problems will be reported as warnings. E.g. it is unusual but allowed to have zero length messages.
- Notes will be displayed when harmless but remarkable behaviour occurs. E.g. MPI_Type_commit might be used twice to commit a datatype that has already been committed.
- The MPI calls will be traced on each node throughout the whole application.
- When detecting a deadlock the last few calls (as configured by the user) can be traced back on each node.
- The output will be written to standard output or to a file. The format will be human readable.
- The output log is produced by the additional process running on the debug server.

1.10.2. Configuration

The following configurations can be set via environment variables:

- Three levels of debug mode:
  1. errors;
  2. errors and warnings;
  3. errors, warnings and remarks.
- Tracing, i.e. whether MPI calls shall be traced or not.
- Maximum message time, i.e. how long a processing unit is allowed to stay idle before it is reported as pending.
- Maximum number of MPI calls that can be traced back in the case of a dead-lock.

1.11. FURTHER GUIDELINES FOR DEVELOPMENT

- Comment your code!!! Use javadoc comments for documentation. They serve also as base for the documentation generated by doxygen.
- Use descriptive names for classes, methods, variables, etc!
- Do not use magic numbers! This makes horrible and unreadable code. Use constants or enums instead.
- TODO: examples

RODUCT TESTING

MARMOT was tested with

- simple test programs,
- applications from WP1,
- other applications and
- benchmarks (for example NAS Parallel Benchmarks)

on different platforms, using different compilers and MPI implementations, for example on

- Linux Clusters with IA32/IA64 processors,
- IBM Regatta and
- NEC SX systems.

On the CrossGrid testbed, MPI jobs may be submitted using Migrating Desktop or using jdl or rsl scripts. MARMOT's log files reveal if the tool is able to find the errors that were deliberately put into the code or not. Sometimes the MPI implementation itself gives an error message, but there are also many cases where the MPI implementation does not detect any error and simply gives wrong results or is aborted for unknown reasons.

- MARMOT's source code contains two subdirectories TEST_C and TEST_F with simple MPI test programs written in C/Fortran. Some of these programs are correct, some of these programs are erroneous to evoke MARMOT's warnings. Users may play with these programs to get a taste of MARMOT.
- For the Crossgrid tutorial there is a special program TEST_C/cg-tutorial-marmot-exercise.c.
- Any application from WP 1 might be used but will not evoke enough warnings from MARMOT to really demonstrate its functionality.
- After installation, the directories /opt/cg/bin/cg-wp2.2-marmot/C and /opt/cg/bin/cg-wp2.2-marmot/F and /opt/cg/bin/cg-wp2.2-marmot-g2/C and /opt/cg/bin/cg-wp2.2-marmot-g2/F contain some example binaries, see section 1.1 about the files to be installed.
- To run the application with MARMOT, one has to add an additional process working as debug server, i.e. one needs (n+1) instead of n processes

  ```
  $ mpirun -np (n+1) foo
  ```

  MARMOT's output is sent to stderr.
- For the use on the testbed, the simple test program basic is installed in /opt/cg/bin/cg-wp2.2-marmot/C. It only performs MPI_Init and MPI_Finalize. Use for example a jdl file like the following basic.jdl:

  ```
  Executable           = "basic";
  JobType              = "MPICH";
  NodeNumber           = 3;
  VirtualOrganisation  = "cg";
  StdOutput            = "basic.out";
  StdError             = "$HOME/basic.err";
  InputSandbox         = {"basic"};
  OutputSandbox        = {"basic.out","$HOME/basic.err"};
  ```

  to submit your job via Resource Broker

  ```
  $ edg-job-submit basic.jdl
  ```

  You can watch the output with something like

```
$ tail –f basic.err
```

on the CE where the job runs, and later on, you can get the output with edg-job-get-output. Same for mpich-g2 jobs.

- Applications written in C can be compiled accordingly to this example:

```
gcc -I/opt/cg/mpich/include -c basic.c
g++  -o basic  basic.o -L../LIB -ldmpi -lmpo  -L/opt/cg/mpich/lib -lmpich

(with gcc = /opt/gcc-3.2.2/bin/gcc-3.2.2, g++ = /opt/gcc-3.2.2/bin/g++-3.2.2)
```

  It is also possible to use mpicc, in this case it is very important to link the proper version of the libstdc++ (i.e. same version as was used to compile the MARMOT libraries, i.e. specify the correct path for -lstdc++):

```
mpicc -c basic.c
mpicc -o basic basic.o -L../LIB -ldmpi -lmpo  -L/opt/cg/mpich/lib -lmpich \
-L/opt/gcc-3.2.2/lib -lstdc++
```

- Applications written in Fortran can be compiled accordingly to this example:
  (with g77 = /opt/gcc-3.2.2/bin/g77-3.2.2, g++ = /opt/gcc-3.2.2/bin/g++-3.2.2):

```
g77 -I/opt/cg/mpich/include -g -O2 -c basic.f
g++ -I/opt/cg/mpich/include -g -O2 -o basic  basic.o \
 -L../LIB -ldmpi -lfmpo -lmpo \
 -L/opt/cg/mpich/lib -lmpich  \
 -L/opt/gcc-3.2.2/lib/gcc-lib/i686-pc-linux-gnu/3.2.2 \
 -L/opt/gcc-3.2.2/lib/gcc-lib/i686-pc-linux-gnu/3.2.2/../../.. -lfrtbegin \
 -lg2c -lm -lgcc_s
```

  It is also possible to use mpif77, in this case it is very important to link the proper version of the libstdc++ (i.e. same version as was used to compile the MARMOT libraries, i.e. specify the correct path for -lstdc++):

```
mpif77 -c basic.f
mpif77 -o basic basic.o -L../LIB -ldmpi -lfmpo -lmpo  \
-L/opt/cg/mpich/lib -lmpich \
-L/opt/gcc-3.2.2/lib -lstdc++
```

- See also the USERS_GUIDE and CONFIG-EXAMPLES file in MARMOT’s distribution.

CONTACT INFORMATION AND CREDITS

If you have any questions, suggestions or bug reports, please contact the developers

Bettina Krammer, krammer@hlrs.de, Matthias Müller, mueller@hlrs.de

HLRS

Allmandring 30

D-70565 Stuttgart

Phone: ++49 711 685 8038

Fax: ++49 711 678 7626

THE GPL LICENSE AGREEMENT

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.